Embedding CCSL into Dynamic Logic: A Logical Approach for the Verification of CCSL Specifications

Yuanrui Zhang, Hengyang Wu, Yixiang Chen, Frédéric Mallet

Embedding CCSL into Dynamic Logic: A Logical Approach for the Verification of CCSL Specifications

Yuanrui Zhang[1], Hengyang Wu[1], Yixiang Chen[1], and Frédéric Mallet[2⋆]

[1] MoE Engineering Research Center for Software/Hardware Co-design Technology and Application, East China Normal University, Shanghai 200062, China
[2] Université Cote d'Azur, I3S, CNRS, Inria, 06900 Sophia Antipolis, France

Abstract. The Clock Constraint Specification Language (CCSL) is a clock-based specification language for capturing causal and chronometric constraints between events in Real-Time Embedded Systems (RTESs). Due to the limitations of the existing verification approaches, CCSL lacks a full verification support for 'unsafe CCSL specifications' and a unified proof framework. In this paper, we propose a novel verification approach based on theorem proving and SMT-checking. We firstly build a logic called CCSL Dynamic Logic (CDL), which extends the traditional dynamic logic with 'signals' and 'clock relations' as primitives, and with synchronous execution mechanism for modelling RTESs. Then we propose a sound and relatively complete proof system for CDL to provide the verification support. We show how CDL can be used to capture RTES and verify CCSL specifications by analyzing a simple case study.

1 Introduction

UML/MARTE [1] is an extension of UML dedicated to the modelling and analysis of Real-Time Embedded Systems (RTESs). Its time model relies on so-called clocks to identify control and observation points in the UML model. These clocks can be used to specify how the system behaves. The Clock Constraint Specification Language (CCSL) [2, 3] is a formal declarative language defined in an annex of MARTE to specify the expected behaviour of the model. Given a system model (or a concrete implementation) and a CCSL specification, the question to answer is whether the system can only perform behaviors that are accepted by the CCSL specification [4]. When a CCSL specification can be encoded as a finite transition system, it is called 'safe' [5], then the verification task mainly consists in making reachability analysis on the product of the system and the CCSL specification. Most recently SMT encoding of CCSL [6] proved to be a promising way to verify unsafe CCSL specification, however, there is no proof environment available so far for reasoning on general specifications.

⋆ Corresponding Author. This work was partly funded by the French Government, through program #ANR-11-LABX-0031-01

In this paper, we propose a novel approach for the verification of CCSL, which is based on the combination of theorem proving and SMT-checking. To capture both the system model and the CCSL specification, we choose dynamic logic [7], since it contains both dynamic program and static logic as its primitives. We propose a variation of dynamic logic, called 'CCSL Dynamic Logic' (CDL), which extends the traditional First-Order Dynamic Logic (FODL) [8] with 'signal' and 'CCSL clock relations' as primitives in its syntax. CDL also supports synchronous events in order to capture synchronous system models [9]. We propose a sound and relatively complete proof system for CDL in order to verify CDL formulas in a modular way.

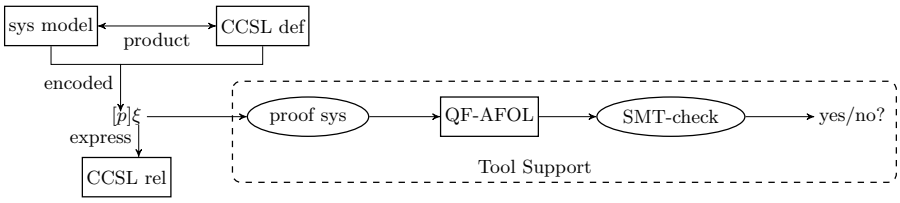


Fig. 1. Verification framework of CDL

Our approach for verification of CCSL specifications can be illustrated in a verification framework given in Fig. 1. The verification task can be captured as a CDL formula of the form $[p]\xi$, where part of the CCSL specification, called 'clock relations' (will be introduced in Sect. 2), are expressed by a formula ξ, and the product of the system model and 'clock definitions' (the other part of the CCSL specification) can be captured by a program of CDL p. In CDL, a formula $[p]\xi$ can be transformed into Quantifier-Free, Arithmetical First-Order Logic (QF-AFOL) formulas through a deduction procedure in the proof system of CDL. Then the validity of these formulas can be handled by an SMT-checking procedure in an efficient way [10], and according to which the verification result is obtained. With CDL, CCSL specifications can be verified in a unified proof framework, provided with strong tool support, e.g. Isabelle [11] and Coq [12].

The rest of this paper is organized as follows: Sect. 2 gives a general introduction to CCSL and FODL. Sect. 3 introduces the syntax and semantics of CDL. In Sect. 4, we propose the proof system for CDL. In Sect.5, we give a simple case study to show how CDL can express and verify CCSL verification problems. Sect. 6 introduces the related works, and Sect. 7 concludes this paper and discusses about future work.

2 Preliminaries of CCSL and FODL

We present the syntax and semantics of CCSL based on [4, 13]. In CCSL, a logical clock actually models a sequence of occurrences of a signal in synchronous models [14]. A logical clock c is defined as an infinite sequence of instants $(c^i)_{i\in\mathbb{N}^+}$,

where each c^i can be 'tick' or 'idle', representing that the signal associated to c occurs or not at a discrete time i. $\mathbb{N}^+$ is the set of natural numbers. Clock relations describe binary relationships between clocks. The syntax of clock relations is defined by:

$$Rel ::= c_1 \subseteq c_2 \mid c_1 \# c_2 \mid c_1 \prec c_2 \mid c_1 \preceq c_2,$$

where c_1, c_2 are arbitrary clocks. We use $\mathcal{C}$ to denote a finite set of clocks. A schedule $\sigma : \mathbb{N} \to \mathcal{P}(\mathcal{C})$ is a finite or infinite sequence of clock ticks, $\mathbb{N} = \mathbb{N}^+ \cup \{0\}$. It gives a global view of how each clock ticks at each instant. For any $i \in \mathbb{N}^+$, $\sigma(i) = \{c \mid c \in \mathcal{C} \wedge c^i = tick\}$. $\sigma(0) = \emptyset$ indicates the beginning of the sequence where no clock ticks. $\mathcal{X}_\sigma : \mathcal{C} \times \mathbb{N}^+ \to \mathbb{N}$ keeps track of the number of ticks for each clock. $\mathcal{X}_\sigma(c, i) = |\{j \mid j \in \mathbb{N}^+, j \leq i, c \in \sigma(j)\}|$ is called a configuration of clock c at time i. The semantics of clock relations is defined as items 1-4 in Table 1. 'Subclock' says that c_1 can only tick if c_2 ticks; 'Exclusion' means that c_1, c_2 can not tick at the same instant; 'Precedence' means that c_1 always ticks faster than c_2; 'Causality' expresses that c_1 ticks not slower than c_2.

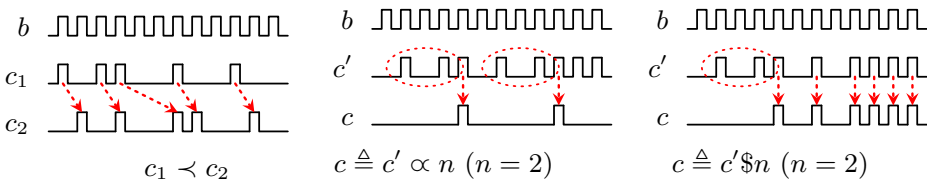
For example, the leftmost figure of Fig. 2 shows a possible schedule σ for clock relation $c_1 \prec c_2$, where clock

$$b = tick\ tick\ tick\ tick\ tick\ tick\ tick\ tick\ tick\ tick\ tick\ tick\ tick\ ...,$$
$$c_1 = tick\ idle\ tick\ tick\ idle\ idle\ tick\ idle\ idle\ tick\ idle\ idle\ ...,$$
$$c_2 = idle\ tick\ idle\ tick\ idle\ idle\ tick\ tick\ idle\ idle\ tick\ idle\ ....$$

b is a based clock representing the minimal granularity of time. Schedule

$$\sigma = \emptyset\{c_1\}\{c_2\}\{c_1\}\{c_1, c_2\}\emptyset\emptyset\{c_1, c_2\}\{c_2\}\emptyset\{c_1\}\{c_2\}\emptyset....$$

$\mathcal{X}_\sigma(c_1, 1) = 1$, $\mathcal{X}_\sigma(c_1, 2) = 1$, $\mathcal{X}_\sigma(c_1, 3) = 2$. $\mathcal{X}_\sigma(c_2, 1) = 0$, $\mathcal{X}_\sigma(c_2, 2) = 1$.



$c_1 \prec c_2$ $\quad c \triangleq c' \propto n\ (n = 2)$ $\quad c \triangleq c'\$n\ (n = 2)$

Fig. 2. A possible schedule for selected clock constraints

Clock definition enhances the expressiveness of CCSL by allowing new clocks to be defined using different clock expressions. A clock definition is of the form: $Cdf ::= c \triangleq E$ where E is a clock expression defined by the following grammar:

$$E ::= c_1 + c_2 \mid c_1 * c_2 \mid c_1 \blacktriangleright c_2 \mid c_1 \rhd c_2 \mid c_1 \curvearrowright c_2 \mid c \propto n \mid c\$n \mid c_1 \vee c_2 \mid c_1 \wedge c_2.$$

c_1, c_2 are arbitary clocks. $n \geq 1$. The semantics of clock definitions are defined as items 5-13 in Table 1. 'Union' defines the clock that ticks iff either c_1 or c_2 ticks;

'Intersection' defines the clock that ticks whenever both c_1 and c_2 tick; '(Strict) Sample' defines the clock that (strictly) samples c_1 based on c_2; 'Interruption' defines the clock that ticks as c_1 until c_2 ticks; 'Periodicity' defines the clock that ticks every n ticks of clock c'; 'Delay' defines the clock that ticks when c' ticks but is delayed for n ticks of c'. 'Infimum' ('Supremum') defines the slowest (fastest) clock that is faster (slower) than both c_1 and c_2.

e.g., Fig. 2 shows a possible schedule of clock definitions $c \triangleq c' \propto n$ and $c \triangleq c'\$n$ (when $n = 2$), which are used in the case study we give in Sect. 5.

Table 1. Semantics of CCSL

1.	$\sigma \vDash_{ccsl} c_1 \subseteq c_2$	iff $\forall i \in \mathbb{N}^+.c_1 \in \sigma(n) \rightarrow c_2 \in \sigma(n)$	(Subclock)
2.	$\sigma \vDash_{ccsl} c_1 \# c_2$	iff $\forall i \in \mathbb{N}^+.c_1 \notin \sigma(i) \vee c_2 \notin \sigma(i)$	(Exclusion)
3.	$\sigma \vDash_{ccsl} c_1 \prec c_2$	iff $\forall i \in \mathbb{N}^+.(\mathcal{X}_\sigma(c_1,i) = 0 \wedge \mathcal{X}_\sigma(c_2,i) = 0) \vee \mathcal{X}_\sigma(c_1,i) > \mathcal{X}_\sigma(c_2,i)$	(Precedence)
4.	$\sigma \vDash_{ccsl} c_1 \preceq c_2$	iff $\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c_1,i) \geq \mathcal{X}_\sigma(c_2,i)$	(Causality)
5.	$\sigma \vDash_{ccsl} c \triangleq c_1 + c_2$	iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \vee c_2 \in \sigma(i))$	(Union)
6.	$\sigma \vDash_{ccsl} c \triangleq c_1 * c_2$	iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \wedge c_2 \in \sigma(i))$	(Intersection)
7.	$\sigma \vDash_{ccsl} c \triangleq c_1 \blacktriangleright c_2$	iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_2 \in \sigma(i) \wedge (\exists 0 < j < i)(\forall j \leq k < i).c_1 \in \sigma(j) \wedge c_2 \notin \sigma(k))$	(Strict Sample)
8.	$\sigma \vDash_{ccsl} c \triangleq c_1 \rhd c_2$	iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_2 \in \sigma(i) \wedge (\exists 0 < j \leq i)(\forall j \leq k < i).c_1 \in \sigma(j) \wedge c_2 \notin \sigma(k))$	(Sample)
9.	$\sigma \vDash_{ccsl} c \triangleq c_1 \curvearrowright c_2$	iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \wedge (\forall 0 < j \leq i).c_2 \notin \sigma(j))$	(Interruption)
10.	$\sigma \vDash_{ccsl} c \triangleq c' \propto n$	iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c' \in \sigma(i) \wedge \exists m \in \mathbb{N}^+.\mathcal{X}_\sigma(c',i) = m \cdot (n+1))$	(Periodicity)
11.	$\sigma \vDash_{ccsl} c \triangleq c'\$n$	iff $\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c,i) = max(\mathcal{X}_\sigma(c',i) - n, 0)$	(Delay)
12.	$\sigma \vDash_{ccsl} c \triangleq c_1 \wedge c_2$	iff $\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c,i) = max(\mathcal{X}_\sigma(c_1,i), \mathcal{X}_\sigma(c_2,i))$	(Infimum)
13.	$\sigma \vDash_{ccsl} c \triangleq c_1 \vee c_2$	iff $\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c,i) = min(\mathcal{X}_\sigma(c_1,i), \mathcal{X}_\sigma(c_2,i))$	(Supremum)

A CCSL specification is a conjunction of clock relations and clock definitions, denoted as a triple $SP ::= \langle \mathcal{C}, \widetilde{Cdf}, \widetilde{Rel} \rangle$, where $\mathcal{C}$ is the set of clocks. $\widetilde{Cdf}$ is a set of clock definitions and $\widetilde{Rel}$ is a set of clock relations. $\sigma \vDash_{ccsl} \langle \mathcal{C}, \widetilde{Cdf}, \widetilde{Rel} \rangle$ is defined s.t. $\sigma \vDash_{ccsl} Rel$ and $\sigma \vDash_{ccsl} Cdf$ hold for all $Rel \in \widetilde{Rel}$ and $Cdf \in \widetilde{Cdf}$.

FODL is an extension of propositional dynamic logic with assignment $x := e$ and testing $P?$ in its program model. The FODL we present here is based on [7]. The program of FODL is a regular program, defined as follows:

$$p ::= x := e \mid P? \mid p; p \mid p \cup p \mid p^*,$$

where e is an arithmetical expression. $P?$ means at current state, P is true. $p; q$ means the program first executes p, and after p terminates, it executes q. $p \cup q$ means the program either executes p, or executes q, it is a non-deterministic choice. p^* means the program executes p for a finite number of times. An FODL formula is defined as follows:

$$\phi ::= tt \mid e \leq e \mid [p]\phi \mid \neg\phi \mid \phi \wedge \phi \mid \forall x.\phi,$$

where tt is the boolean true, $\leq$ represents the 'less than' relation in number theory. $[p]\phi$ is the dynamic formula, meaning after all executions of program p, formula ϕ holds.

The semantics of FODL is based on Kripke structure [7]. A Kripke structure is a pair (S, val) where S is a set of states, val is a function that interprets a logic to data structures on S.

In FODL, val interprets a regular program as a set of state pairs (s, s') and interprets a formula as a set of states. Intuitively, each pair $(s, s') \in val(p)$ means that starting from state s, after execution of p, the program may terminate at state s'. Each state $s \in val([p]\phi)$ means that for all pairs $(s, s') \in val(p)$, s' satisfies ϕ. For a formal definition of the semantics of FODL, refer to [7].

The deductive system for FODL is sound and relatively complete. Except for the rule for atomic program '$x := e$', all rules can be found in Table 3, 4 below, as a part of CDL proof system. Refer to [7] for more details.

3 Syntax and Semantics of CDL

CDL enriches the traditional FODL with a synchronous program model that contains 'signal' as a primitive, and 'clock relation' as an ingredient of logic formulas. We first give the syntax of the CDL program model and the CDL formula, and then define their semantics.

3.1 The Syntax of CDL

Syntax of Synchronous Event Programs. CCSL essentially describes the logical and chronometrical constraints between signals in synchronous models, where the time model is discrete and at each time, several signals can be triggered simultaneously. To capture CCSL constraints in dynamic logic, we need to introduce the synchronous execution mechanism in the regular program of FODL. Synchronous systems often involve infinite executions, thus to support it we also import 'infinite loop'. The program after enriched turns out to be an 'omega program', with the support of synchronous mechanism. We call it 'Synchronous Event Program' (SEP).

Definition 1 (Syntax of SEP). *The syntax of SEP is based on the regular program of FODL, defined as follows:*

$$p ::= \varepsilon \mid \alpha \mid P?\alpha \mid p;p \mid p \cup p \mid p^* \mid p^\omega,$$

where α is a combinational event, defined as:

$$\alpha ::= \epsilon \mid Cmb,$$
$$Cmb ::= c \mid x := e \mid (Cmb|Cmb).$$

Arithmetical expression e, testing condition for signals ϱ and testing condition P are defined as follows:

$$e ::= x \mid n \mid e + e \mid e - e \mid n \cdot e \mid e/n,$$
$$P ::= tt \mid e \leq e \mid \neg P \mid P \wedge P.$$

ε represent an 'empty program', it does nothing nor consumes time. A combinational event α consumes a unit of time, it consists of an 'idle event' ϵ, or several signals or assignments that occur simultaneously. An idle event ϵ does nothing but waits for a unit of time. Several signals and assignments can be composed by operator '|'. A signal[3] c in an SEP represents that its corresponding clock (with the same name c) ticks at current time. Since CCSL constraints only captures the logical relationships between signals which are not related to the value of signals, we only consider 'pure signals' (signals without values) in SEP. e is a Presburger arithmetic expression. In e, $n \in \mathbb{Z}$ is an integer number, $+, -, \cdot, /$ are the addition, subtraction, multiplication and division signs respectively.

$P?\alpha$ is a testing event, it means that if condition P is true, event α proceeds, otherwise the program causes a deadlock. In SEP, testing $P?$ must combines with an event α, because $P?$ does not consume time. P can be expressed with a QF-AFOL formula, where tt represents the boolean true, $\leq$ represents the 'less than' relation between two integers. Operator $;, \cup, *$ are defined just as in FODL [7]. ω represents the infinite loop. p^ω means that program p executes for infinite number of times and never terminates.

e.g., program $f = 1?\alpha_2; p^*$ where $p ::= n = 0 \wedge f = 0?\alpha_3 \cup n > 0 \wedge f = 0?\alpha_4$ firstly executes α_2 if $f = 1$ holds, then it executes program p for finite number of times. In p, it either executes α_3 (if $n = 0 \wedge f = 0$ holds), or executes α_4 (if $n > 0 \wedge f = 0$ holds).

The precedence of operators are listed as follows from the highest to the lowest: ω, $*$, ;, $\cup$. We stipulate that ; is right-associative, $\cup$ is left-associative. e.g., program $\alpha_1 \cup p_1; p_2; p_3^\bullet \cup P_1?\alpha_2^\bullet \cup P_2?\alpha_3$ means $(((\alpha_1 \cup p_1; (p_2; p_3^\bullet)) \cup P_1?\alpha_2^\bullet) \cup P_2?\alpha_3)$.

As in synchronous models (e.g., Esterel [14]), we do not allow two signals with the same name triggered at the same time. e.g. event $(c|c)$. For simplification, we also do not allow two assignments with the same target variable executing simultaneously, e.g. event $(x := 5|x := y + 1)$.

Syntax of CDL. In CDL formula, we need to introduce a special kind of variable which is related to clock. These variables help record the 'information' of each clock at current time, just as the roles the schedule σ and the configuration $\mathcal{X}_\sigma$ play in CCSL.

Definition 2 (Clock Related Variables). *For each clock $c \in \mathcal{C}$, we define two variables related to it: c^n, c^s. Variable c^n is of type $\mathbb{N}$, it records the number of times the clock has ticked at current time. Variable c^s is of type $\{0, 1\}$, and it records the status of the clock (1 for present and 0 for absent) at current time.*

Given a clock set $\mathcal{C}$, we denote the set of variables related to $\mathcal{C}$ as $Var(\mathcal{C})$.

[3] In SEP, for convenience, we use the same name 'c' to represent the signal corresponding to clock c, which should not cause any ambiguities. Sometimes we also say a signal c in p 'a clock c in p'.

Definition 3 (Syntax of CDL Formula). *The CDL formula ϕ is defined as:*

$$\phi ::= tt \mid E \leq E \mid [p]\xi \mid [p]\phi \mid \neg\phi \mid \phi \wedge \phi \mid \forall x.\phi$$

where

$$\xi ::= Rel \mid \curlywedge (Rel_1, ..., Rel_n),$$
$$E ::= x \mid c^n \mid c^s \mid n \mid E + E \mid E \cdot E.$$

$E \leq E$ is an atomic AFOL formula. E is an integer arithmetic expression. Different from e, it also includes clock-related variable c^n, c^s, and multiplication between variables. $x \in Var$. $[p]\xi$ is a dynamic formula, where p is an SEP. $[p]\xi$ is the dynamic formula special in CDL, it means that all execution paths of program p satisfies ξ. $\curlywedge(Rel_1, ..., Rel_n)$ represents the conjunction of clock relations $Rel_1, ..., Rel_n$, we define $\sigma \vDash_{ccsl} \curlywedge(Rel_1, ..., Rel_n)$ iff $\sigma \vDash_{ccsl} Rel_1$,...,$\sigma \vDash_{ccsl} Rel_n$. In order to express the negation of $[p]\xi$ in CDL, we also import the negation $\sim$ and the disjunction $\curlyvee$ of clock relations: (i) $\sigma \vDash_{ccsl} \sim cr$ iff $\sigma \nvDash_{ccsl} cr$, (ii) $\sigma \vDash_{ccsl} \curlyvee(cr_1, ..., cr_n)$ iff $\sigma \vDash_{ccsl} \sim \curlywedge(\sim cr_1, ..., \sim cr_n)$, where $cr, cr_i \in \{Rel_i, \sim Rel_i\}(1 \leq i \leq n)$. $[p]\phi$ is the dynamic formula in FODL, meaning that after all executions of p, formula ϕ is satisfied.

We often call ξ or $\sim \xi$ "path formulas", denoted by π. Other arithmetic expressions, relations, and logic expressions, e.g., $E - E$, E/E, $E = E$, $E < E$, ff, $\langle p \rangle \sim \xi$, $\langle p \rangle \phi$, $\phi \vee \phi$, $\phi \rightarrow \phi$, $\exists x.\phi$, etc, can be expressed using the formulas given above. e.g., $\langle p \rangle \sim \xi$ can be expressed as $\neg[p]\xi$, $E_1 - E_2$, E_1/E_2 can be expressed as $\exists x.(E_2 + x = E_1)$, $\exists x.(x \cdot E_2 = E_1)$ respectively.

In FODL, given a formula ϕ, a variable whose value changes with the execution of a program is called a 'dynamic variable' [7] of the formula ϕ. Here in CDL, for convenience sake, any clock-related variable c^s, c^n is defined as a dynamic variable. As we will see in Def. 5(ii), they can be seen 'changed' after the execution of any event at current time. Any general variable that appears on the left side of an assignment is defined as a dynamic variable as well. Variables which are not dynamic variables are called 'static variables'. e.g., the set of dynamic variables of formula $z = 5 \rightarrow [(c_1|x := y{+}1); c_2]c_1 \preceq c_3$ is $\{x, c_1^n, c_1^s, c_2^n, c_2^s, c_3^n, c_3^s\}$, where c_3^n, c_3^s can be seen as 'changed' after the the set of static variables is $\{y, z\}$.

Like in FODL, we say a variable x is 'bound' in ϕ iff: 1. x is in the scope of the effect of some quantifier $\forall x$, or 2. x is in the scope of the effect of some event α which has x on the left side of an assignment of the form $x := e$. A variable is not bound in ϕ is called 'free'. e.g., in formula $\phi ::= (x = 1 \wedge z = 2 \wedge \exists z.x = z) \rightarrow [(x := z + 1|c|y := 1); x := y + 1]x > z$, the first and second variable x is free, while the third one (in expression '$x > z$') is bounded by the assignment '$x := y + 1$'.

Given a formula ϕ, a substitution $\phi[E/x]$ in CDL replaces all the free occurrences of variable x with expression E (of the same type). Given a formula multi-set Γ, $\Gamma[E/x]$ means to carry out the substitution $\phi[E/x]$ for each formula ϕ in Γ. Given two vectors $(E_1, ..., E_n), (x_1, ..., x_n)$, $\phi[E_1, ..., E_n/x_1, ..., x_n]$ is the shorthand of $\phi[E_1/x_1][E_2/x_2]...[E_n/x_n]$. A substitution is admissible with respect to a formula ϕ if there are no variables x, y such that y is in E, and after

the replacement $\phi[E/x]$, y is bound in ϕ. e.g., in the formula ϕ given above, $\phi[z+1/z] = (x = 1 \wedge z+1 = 2 \wedge \exists z.x = z) \rightarrow [(x := (z+1)+1|c|y := 1); x := y+1]x > z+1$ is admissible, while $\phi[x+1/z] = (x = 1 \wedge x+1 = 2 \wedge \exists z.x = z) \rightarrow [(x := (x+1)+1|c|y := 1); x := y+1]x > x+1$ is not admissible. Intuitively, in $\phi[x+1/z]$, it is about to prove $x > x+1$ which is generally not true. In the rest of paper, unless we specially point out, all substitutions we discuss are admissible.

3.2 The Semantics of CDL

In the Kripke structure (introduced in Sect.2) of CDL, *val* interprets a program as a set of traces on S and a logic formula as a set of states. A trace tr is a finite or infinite sequence of states. Given a finite trace $tr_1 = s_1 s_2 ... s_n$ and a (possibly infinite) trace $tr_2 = u_1 u_2 ... u_n ...$, we define: $tr_1 \cdot tr_2 ::= s_1 s_2 ... s_n u_2 u_3 ...$ if $s_n = u_1$. Given any tr_1, tr_2, we define $tr_1 \circ tr_2 ::= \begin{cases} tr_1 \cdot tr_2, & \text{if } tr_1 \text{ is finite} \\ tr_1, & \text{otherwise} \end{cases}$. Given two sets of traces S_1, S_2, $S_1 \circ S_2$ is defined as $\{tr_1 \circ tr_2 \mid tr_1 \in S_1, tr_2 \in S_2\}$. Let $tr(i)$ denotes the i^{th} element of trace tr, $i \geq 0$; tr_b denotes the first element of trace tr, $tr_b = tr(0)$. Let tr_e denotes the last element of trace tr, provided that tr is a finite trace.

In CDL, we assume an interpretation which interprets arithmetical operators '$+, -, \cdot, /$' and relation '$\leq$' as their usual meanings in the traditional number theory, and interprets relations '$\subseteq, \prec, \preceq, \#$' as their corresponding clock relations in CCSL. Next we first define the concept of 'state' in CDL.

Definition 4 (State and Evaluation in CDL). *A state s in CDL is a total function defined as follows:*

(i) s maps each variable c^n in $Var(\mathcal{C})$ to a value in domain $\mathbb{N}$.
(ii) s maps each variable c^s in $Var(\mathcal{C})$ to a value in domain $\{0, 1\}$.
(iii) s maps each variable x in Var to a value in domain $\mathbb{Z}$.

Given an expression E and a state s, an evaluation $Eval_s(E)$ is defined as:

(i) If $E = a$, where $a \in \{x, c^n, c^s\}$, then $Eval_s(a) ::= s(a)$.
(ii) If $E = n$, then $Eval_s(n) ::= n$.
(iii) If $E = f(E_1, E_2)$, where $f \in \{+, \cdot\}$, then $Eval_s(E) ::= f(Eval_s(E_1), Eval_s(E_2))$.

e.g., given a state $s ::= \{x \mapsto 9, c^n \mapsto 2, (c')^s \mapsto 0, ...\}$, there is $Eval_s(2) = 2$, $Eval_s(x) = 9$, $Eval_s(x + c^n) = Eval_s(x) + Eval_s(c^n) = 11$.

Semantics of SEP. Different from traditional FODL, the semantics of SEP is based on traces, since our CDL contains a path formula π which is satisfied by a program trace.

Definition 5 (Semantics of SEP). *Given a Kripke structure (S, val), for any SEP p, let $\mathcal{C}$ be a finite set of clocks, the semantics of SEP is given as follows:*

(i) $val(\varepsilon) := S$, *S is the set of all traces of length 1.*
$val(\alpha) := \{ss' \mid s, s' \in S$; *for each clock* $c \in \alpha, s'(c^s) = 1 \wedge s'(c^n) = s(c^n) + 1$;

(ii) *for other clock* $d \in \mathcal{C}, s'(d^n) = s(d^n) \wedge s'(d^s) = 0$; *for each* $x := e$ *in* α, $s'(x) = Eval_s(e)$; *for other* $x \in Var, s'(x) = s(x)\}$.

(iii) $val(P?\alpha) ::= \{ss' \mid s \in val(P), ss' \in val(\alpha)\}$.

(iv) $val(p; q) ::= val(p) \circ val(q)$.

(v) $val(p \cup q) ::= val(p) \cup val(q)$.

(vi) $val(p^*) ::= \bigcup_{n \geq 0} val^n(p)$, *where* $val^n(p) = \underbrace{val(p) \circ ... \circ val(p)}_{n}$, $val^0(p) = S$.

(vii) $val(p^\omega) ::= \underbrace{val(p) \circ val(p) \circ ....}_{\infty}$

Note that ε defines a set of traces of length 1, so $val(p; \varepsilon) = val(\varepsilon; p) = val(p)$, which means that ε can be taken as a unit element of operator ;. Event α defines a transition from a state s to a state s'. In s', for each clock c in α, the variable c^n that records the number of ticks is added by 1 and the variable c^s is set to 1, indicating at current time, clock c is emitted. For each clock d not in α, its variable d^n in s' is kept the same while d^s is set to 0. For any assignment $x := e$ in α, the value of x in s' is set to the value of expression e in state s, while other variables in both s and s' are kept the same. Traces satisfying $P?\alpha$ are exactly those traces satisfying p adding that their beginning states must satisfy P.

e.g., let $\alpha = (c|x := x+1)$, $P = x > 1$, $\mathcal{C} = \{c, c'\}$, $Var = \{x, y\}$, if $s = \{x \mapsto 0, y \mapsto 0, c^n \mapsto 0, c^s \mapsto 0, {c'}^n \mapsto 0, {c'}^s \mapsto 0\}$, $s' = \{x \mapsto 1, y \mapsto 0, c^n \mapsto 1, c^s \mapsto 1, {c'}^n \mapsto 0, {c'}^s \mapsto 0\}$, then trace $ss' \in val(\alpha)$. If $u = \{x \mapsto 2, y \mapsto 0, c^n \mapsto 1, c^s \mapsto 1, {c'}^n \mapsto 0, {c'}^s \mapsto 1\}$, $u' = \{x \mapsto 3, y \mapsto 0, c^n \mapsto 2, c^s \mapsto 1, {c'}^n \mapsto 0, {c'}^s \mapsto 0\}$, then trace $uu' \in val(P?\alpha)$.

The semantics of $p; q, p \cup q$, p^* are directly inherited from the traditional FODL [7]. The traces of program p^ω consists of all infinite traces of the form $tr_1 \circ tr_2 ...$ where each $tr_i \in val(p)$ is finite ($i \in \mathbb{N}^+$), or of the form $tr_1 \circ tr_2 \circ ... \circ tr_n$, where $n \geq 1$, $tr_1, ..., tr_{n-1} \in val(p)$ is finite, but $tr_n \in val(p)$ is infinite. e.g., suppose $val(p) = \{s_1 s_2\}$, $val(q) = \{u_1 u_2, t_1 t_2\}$ where $s_2 = u_1$, $s_2 \neq t_1$, then $val(p; q) = \{s_1 s_2 u_2\}$, $val(p \cup q) = \{s_1 s_2, u_1 u_2, t_1 t_2\}$, $val(p^*) = \{\varepsilon, s_1 s_2, s_1 s_2 s_1 s_2, ..., \underbrace{s_1 s_2 s_1 s_2 ... s_1 s_2}_{2n}, ...\}$ $(n \geq 1)$, $val(p^\omega) = val(p^*) \cup \{\underbrace{s_1 s_2 s_1 s_2 ... s_1 s_2 ...}_{\infty}\}$.

Semantics of CDL. For each trace tr, we can actually build a corresponding schedule σ^{tr} s.t. for all clock $c \in \mathcal{C}$ and $i \in \mathbb{N}^+$, there is: 1. $tr(i)(c^n) = \mathcal{X}_\sigma(c, i)$. 2. $tr(i)(c^s) = 1$ iff $c \in \sigma^{tr}(i)$. In this way, we can actually define $tr \vDash_{ccsl} X$ given a clock relation or definition X: $tr \vDash_{ccsl} X$ iff $\sigma^{tr} \vDash_{ccsl} X$. Note that we do not require any relationships between $tr(0)$ and $\sigma(0)$.

e.g., consider the trace ss' discussed above, we have a schedule $\sigma^{ss'}$ defined as: $\sigma^{ss'} ::= \emptyset\{c\}$. So $\mathcal{X}_{\sigma^{ss'}}(c, 0) = \mathcal{X}_{\sigma^{ss'}}(c', 0) = 0$, $\mathcal{X}_{\sigma^{ss'}}(c, 1) = 1$, $\mathcal{X}_{\sigma^{ss'}}(c', 1) = 0$.

Definition 6 (Semantics of CDL Formula). *Given a Kripke structure* (S, val)*, the semantics of CDL formula is given as follows:*

(i) $val(tt) ::= S$.
(ii) $val(E \leq E') ::= \{s \mid Eval_s(E) \leq Eval_s(E')\}$.
(iii) $val([p]\xi) ::= \{s \mid$ *for all* tr *s.t.* $s = tr_b$ *and* $tr \in val(p), tr \vDash_{ccsl} \xi\}$.
(iv) $val([p]\phi) ::= \{s \mid$ *for all finite* $tr \in val(p)$ *s.t.* $tr_b = s, tr_e \in val(\phi)\}$.
(v) $val(\neg\phi) ::= \{s \mid s \notin val(\phi)\}$.
(vi) $val(\phi \wedge \varphi) ::= val(\phi) \cap val(\varphi)$.
(vii) $val(\forall x.\phi) ::= \{s \mid$ *for any* $v_0 \in \mathbb{Z}, s \in val(\phi[v_0/x])\}$.

The semantics of CDL formula is based on states. In (iii), a trace satisfying a clock relation is from the second state of the trace due to the definition of $\vDash_{ccsl}$ in Sect. 2. So state s itself is unrelated to ξ. (iv)-(vii) are similar to the definition in FODL [7], except that the semantics of SEP is based on traces. (iv) requires the trace must be finite, indicating that it only matters whether ϕ holds on those states on which program p terminates.

The first two figures in Fig. 3 give an illustration of $[p]\phi$ and $[p]\xi$, where the 'snake arrow' indicates an execution path (could be infinite) of program. Some states are tagged with a formula aside that they satisfy. States and paths are colored red to stress that they satisfy the corresponding formulas (ϕ, ξ).

At last we define the satisfaction relation of the CDL logic. Given a state s and any CDL formula ϕ, the satisfaction relation $s \models_{cdl} \phi$ is defined as: $s \models_{cdl} \phi$ iff $s \in val(\phi)$. If for all state s, $s \models_{cdl} \phi$ holds, then we say ϕ is valid, denoted as $\models_{cdl} \phi$.

4 Proof System of CDL

In this section we propose a proof system, which forms the foundation of the verification of CDL. The proof system provides a modular way of transforming a CDL formula into a QF-AFOL formula. Our proof system is based on that of FODL, which is only for regular program model [7].

A sequent [15] is defined as follows: $\Gamma \Rightarrow \Delta ::= \bigwedge_{\phi\in\Gamma} \phi \rightarrow \bigvee_{\varphi\in\Delta} \varphi$, where Γ, Δ are two finite *multi-sets* of logic formulas. It means that every formula in Γ holds can conclude that at least one of formulas in Δ holds. The conditions when either (both) Σ or (and) Δ is (are) empty set(s) is (are) expressed as follows: 1. $\cdot \Rightarrow \Delta ::= tt \rightarrow \bigvee_{\varphi\in\Delta} \varphi$, 2. $\Gamma \Rightarrow \cdot ::= \bigwedge_{\phi\in\Gamma} \phi \rightarrow ff$, 3. $\cdot \Rightarrow \cdot ::= tt \rightarrow ff$, where we use $\cdot$ to indicate Γ or Δ is empty. A rule in sequent calculus is of the form: $\dfrac{\Gamma_1 \Rightarrow \Delta_1 \quad ... \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}$, which means that if $\Gamma_1 \Rightarrow \Delta_1$, ..., $\Gamma_n \Rightarrow \Delta_n$ are all valid, then $\Gamma \Rightarrow \Delta$ is valid. Each $\Gamma_i \Rightarrow \Delta_i$ in the upper part is called a 'premise', while $\Gamma \Rightarrow \Delta$ in the lower part is called 'conclusion'. We use $\dfrac{\Gamma \Rightarrow \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \phi \Rightarrow \Delta}$ to represent a pair of sequent rules: $\dfrac{\Gamma, \varphi \Rightarrow \Delta}{\Gamma, \phi \Rightarrow \Delta}$ and $\dfrac{\Gamma \Rightarrow \varphi, \Delta}{\Gamma \Rightarrow \phi, \Delta}$, i.e., ϕ, φ can be on both side of the sequent. Sometimes we write $\dfrac{\varphi}{\phi}$ to represent $\dfrac{\Gamma \Rightarrow \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \phi \Rightarrow \Delta}$ if

Γ, Δ can be neglected. We call Γ, Δ the context of formula ϕ in sequent $\Gamma \Rightarrow \phi, \Delta$ or $\Gamma, \phi \Rightarrow \Delta$.

4.1 Proof Rules for CDL

The proof rules of CDL we present are divided into three categories: rules for path formulas π (in Table 2), rules for non-path formulas (in Table 3) and rules of First-Order Logic (FOL) (in Table 4).

In Table 2, rule (π) is for a single event, where we set $\alpha = (c|x := e)$ as an example of combinational events.

Table 2. Rules for path formulas

$$\frac{\begin{array}{c}\Gamma[V'/V], c^n = (c^n)' + 1, c^s = 1, x = e[V'/V], \\ (d_1^n, ..., d_n^n) = ((d_1^n)', ..., (d_n^n)'), (d_1^s, ..., d_n^s) = \underbrace{(0, ..., 0)}_{n} \Rightarrow \\ \hbar(\xi) \Rightarrow \Delta[V'/V]\end{array}}{\Gamma \Rightarrow [\alpha]\xi \Rightarrow \Delta}\ (\pi)$$

where $\alpha = (c|x := e)$, $\{d_1, ..., d_n\} = \mathcal{C} - \mathcal{C}(\alpha)$, $V = \mathcal{V}(\alpha)$, V' is the set of new variables (w.r.t. $\Gamma, [\alpha]\xi, \Delta,$) corresponding to V.

$$\frac{P \to [\alpha]A}{[P?\alpha]A}\ (P?) \qquad \frac{tt}{[\varepsilon]\xi}\ (\pi\varepsilon) \qquad \frac{[p^*]\xi}{[p^\omega]\xi}\ (\pi[\omega]) \qquad \frac{[p]\xi \wedge [p][q]\xi}{[p;q]\xi}\ (\pi[;])$$

where $A \in \{\xi, \phi\}$

$$\frac{[p]\xi \wedge [q]\xi}{[p \cup q]\xi}\ (\pi[\cup]) \qquad \frac{[p;p^*]\xi}{[p^*]\xi}\ (\pi[*]u) \qquad \frac{[p^*][p]\xi}{[p^*]\xi}\ (\pi[*]i)$$

The rule says that for any state s, the conclusion holds at state s, iff there exists a state s' with $ss' \in val((c|x := e))$, s.t. the premise holds at s'. The vector equation $(x_1, ..., x_n) = (e_1, ..., e_n)$ is the shorthand of $x_1 = e_1, ..., x_n = e_n$. $d_1, ..., d_n$ are all clocks not appeared in α. Given a CDL formula ϕ (or an SEP p), let $\mathcal{C}(\phi)$ ($\mathcal{C}(p)$) returns all clocks appeared in ϕ (p), $\mathcal{V}(\phi)$ ($\mathcal{V}(p)$) returns all dynamic variables appeared in ϕ (p). V' is the set of new variables corresponding to V, for each variable $x \in V$, there is a new variable x' with respect to $\Gamma, [\alpha]\xi, \Delta$ corresponding to it. Function $\hbar(\xi)$ maps each relations to an AFOL formula which should hold at state s'. It is defined as follows: for any c_1, c_2, (i) $\hbar(c_1 \subseteq c_2) ::= c_1^s = 1 \to c_2^s = 1$. (ii) $\hbar(c_1 \# c_2) ::= c_1^s = 0 \vee c_2^s = 0$. (iii) $\hbar(c_1 \prec c_2) ::= (c_1^n = 0 \wedge c_2^n = 0) \vee (c_1^n > c_2^n)$. (iv) $\hbar(c_1 \preceq c_2) ::= c_1^n \geq c_2^n$. (v) $\hbar(\lambda(Rel_1, ..., Rel_n) ::= \bigwedge_{1 \leq i \leq n} \hbar(Rel_i)$.

$(P?)$ is a rule for both path-formulas and non-path formulas. Rule $(P?)$ says that the conclusion at a state is true, iff if P is true, then $[\alpha]A$ is true. In rule $(\pi\varepsilon)$, $tr \vDash_{ccsl} \xi$ always holds for trace tr of length 1. Rule $(\pi\omega)$ is based on two

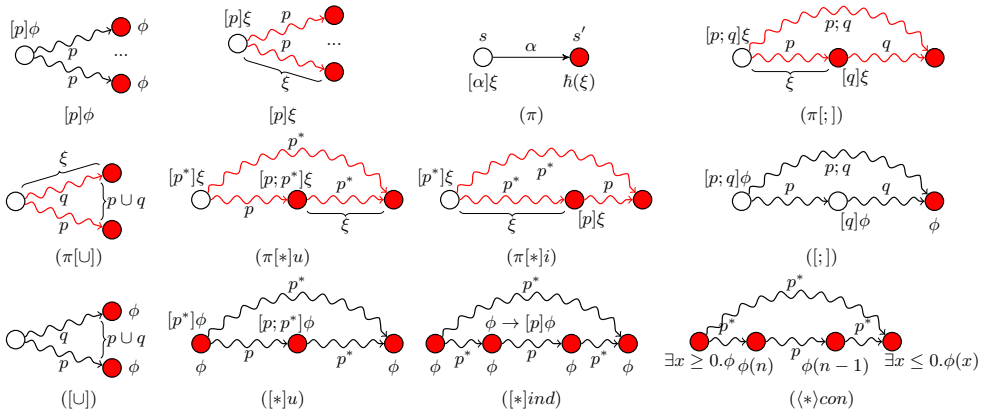


Fig. 3. Graphical illustrations of $[p]\phi$, $[p]\xi$ and some proof rules

facts about clock relation ξ and SEP traces: (i) For any infinite trace $tr \in val(p^\omega)$ and any state s in tr, there exists a finite trace $tr' \in val(p^*)$ that contains s. (ii) For any relation ξ and trace tr, $tr \vDash_{ccsl} \xi$ iff $tr(i) \models_{cdl} \hbar(\xi)$ for any $i \in \mathbb{N}^+$. These two facts can be easily obtained according to Def. 5 and the definition of $\vDash_{ccsl}$ in Table 1. With them not hard to see the premise and conclusion of rule $(\pi\omega)$ are logical equivalent. With rule $(\pi\omega)$ we can reduce the proof case of $[p^\omega]\xi$ to the proof case of $[p^*]\xi$.

$(\pi[;])$, $(\pi[\cup])$, $(\pi[*]u)$, $(\pi[*]i)$ are structure rules for path formulas. $(\pi[;])$ means every trace of $p;q$ satisfies ξ iff every trace of p satisfies ξ, and after p every trace of q satisfies ξ. $(\pi[\cup])$ says every trace of $p \cup q$ satisfies ξ iff every trace of p and q satisfies ξ. Rule $(\pi[*]u)$ unwinds the star operator $*$. It is due to the fact that every trace (whose length ≥ 2) of p^* are the trace of $p;p^*$. $(\pi[*]i)$ states that ξ holds along all paths of any times of repetitions of p, iff after any times of repetitions p , ξ holds along all paths of p. Fig. 3 gives a graphical illustration of rule (π), $(\pi[;])$, $(\pi[\cup])$, $(\pi[*]u)$, $(\pi[*]i)$.

All non-path formula rules in Table 3 except for (ϕ), (ε), (ω) are based on the corresponding structure rules of FODL in [7]. (ϕ) is similar to (π), except that ϕ is kept unchanged in the premise. (ε) is obvious because the traces of ε all have length 1. Rule $([;])$ describes that ϕ holds after $p;q$ iff $[q]\phi$ holds after p. Rule $([\cup])$ says ϕ holds after $p \cup q$ iff ϕ holds after p, and also holds after q. $([*]u)$ means that ϕ holds after any times of repetitions of p, iff ϕ holds at current state, and ϕ holds after $p;p^*$.

$([]gen)$, $(\langle\rangle gen)$ strengthen the conclusions by extending the proposition $\phi \to \varphi$ into dynamic situations. $([]gen)$ $((\langle\rangle gen))$ expresses that if $\phi \to \varphi$ holds under all context of Γ, Δ, after any (some) executions of p, ϕ implies φ. $([*]ind)$ is the mathematical induction by the number of repetitions of program p: to prove ϕ holds after any repetitions (including 0), we need to prove that under any context of Γ, Δ, if ϕ holds, then it also holds after p. $(\langle*\rangle con)$ is from the Harel's convergence rule in [7] where integer x indicates the existing number of

repetitions of p. $([*]i)$ and $(\langle * \rangle i)$ are rules for eliminating the star operator $*$ in practical verification. They can be derived by $([*]ind)$, $(\langle * \rangle con)$ with generalisation $([]gen)$, $(\langle\rangle gen)$ (see [7, 16]). φ is the loop invariant of p. $([*]i)$ says that to prove ϕ holds after any repetitions of p, we need to prove that there exists an invariant φ such that: (i) φ holds at the beginning. (ii) Under any context of Γ, Δ if φ holds, then φ holds after p as well. (iii) Under any context of Γ, Δ, φ implies ϕ. Fig. 3 gives a graphical illustration of rule $([\cup])$, $([;])$, $([*]u)$, $([*]ind)$, $(\langle * \rangle con)$.

Table 3. Rules for non-path formulas

$$\frac{\begin{array}{c}\Gamma[V'/V], c^n = (c^n)' + 1, c^s = 1, x = e[V'/V], \\ (d_1^n, ..., d_n^n) = ((d_1^n)', ..., (d_n^n)'), (d_1^s, ..., d_n^s) = \underbrace{(0, ..., 0)}_{n} \Rightarrow \\ \phi \Rightarrow \Delta[V'/V]\end{array}}{\Gamma \Rightarrow [\alpha]\phi \Rightarrow \Delta} \ (\phi)$$

where $\alpha = (c|x := e)$, $\{d_1, ..., d_n\} = \mathcal{C} - \mathcal{C}(\alpha)$, $V = \mathcal{V}(\alpha)$,

V' is the set of new variables (w.r.t. $\Gamma, [\alpha]\phi, \Delta,$) corresponding to V.

$$\frac{\phi}{[\varepsilon]\phi} \ (\varepsilon) \quad \frac{tt}{[p^\omega]\phi} \ (\omega) \quad \frac{[p][q]\phi}{[p;q]\phi} \ ([;]) \quad \frac{[p]\phi \wedge [q]\phi}{[p \cup q]\phi} \ ([\cup]) \quad \frac{\phi \wedge [p;p^*]\phi}{[p^*]\phi} \ ([*]u)$$

$$\frac{\cdot \Rightarrow \phi \rightarrow \varphi}{\Gamma \Rightarrow [p]\phi \rightarrow [p]\varphi, \Delta} \ ([]gen) \quad \frac{\cdot \Rightarrow \phi \rightarrow \varphi}{\Gamma \Rightarrow \langle p \rangle \phi \rightarrow \langle p \rangle \varphi, \Delta} \ (\langle\rangle gen) \quad \frac{\cdot \Rightarrow \phi \rightarrow [p]\phi}{\Gamma \Rightarrow \phi \rightarrow [p^*]\phi, \Delta} \ ([*]ind)$$

$$\frac{\cdot \Rightarrow \forall x > 0.(\phi(x) \rightarrow \langle p \rangle \phi(x-1))}{\Gamma \Rightarrow \exists x \geq 0.\phi(x) \rightarrow \exists x \leq 0.\langle p^* \rangle \phi(x), \Delta} \ (\langle * \rangle con)$$

$$\frac{\Gamma \Rightarrow \varphi, \Delta \quad \cdot \Rightarrow \varphi \rightarrow [p]\varphi \quad \cdot \Rightarrow \varphi \rightarrow \phi}{\Gamma \Rightarrow [p^*]\phi, \Delta} \ ([*]i)$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \exists x \geq 0.\varphi(x), \Delta \quad \cdot \Rightarrow \forall x > 0.(\varphi(x) \rightarrow \langle p \rangle \varphi(x-1)) \\ \cdot \Rightarrow \exists x \leq 0.\varphi(x) \rightarrow \phi\end{array}}{\Gamma \Rightarrow \langle p^* \rangle \phi, \Delta} \ (\langle * \rangle i)$$

Other FOL rules are listed in Table 4. As indicated in Sect.1, after a QF-AFOL formula is obtained we can adopt SMT-checking procedure to check the validation of it. Since the SMT-checking procedure is independent from the CDL proof system, we propose an 'oracle' rule (o) in our proof system to indicate the termination of the proof. We assume that the validity of this QF-AFOL formula can be SMT-checked in a 'black box', through this oracle rule. Other rules comes from the traditional FOL and we omit the details of them.

Now we define the deduction relation of CDL. For any CDL formula ϕ and a formula multi-sets Φ, $\Phi \vdash_{cdl} \phi$ iff the sequent $\Phi \Rightarrow \phi$ can be derived according

to rules in Table 2, 3, 4. If Φ is empty, we also write $\vdash_{cdl} \phi$. As a variation of dynamic logic, the soundness and relative completeness of proof system $\vdash_{cdl}$ can be analyzed in a similar way as those of FODL in [7, 8]. For the soundness, above we have explained the intuitive meaning of each rule and their relations to the corresponding rules in FODL. For the relative completeness, intuitively, to prove it we show that each formula of form $[p]\xi$ can be transformed into an AFOL formula by applying the rules of the CDL proof system, which is similar for the formula $[p]\phi$ in FODL. Due to space limit, we omit the complete proof.

Table 4. Rules of first order logic

$$\frac{\models_{cdl} \bigwedge_{\phi\in\Gamma}\phi \to \bigvee_{\varphi\in\Delta}\varphi}{\Gamma \Rightarrow \Delta}\ (o) \qquad \frac{}{\Gamma,\phi \Rightarrow \phi,\Delta}\ (ax) \qquad \frac{\Gamma \Rightarrow \phi,\Delta \quad \Gamma,\phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ (cut)$$

$$\frac{\Gamma,\neg\phi \Rightarrow \Delta}{\Gamma \Rightarrow \phi,\Delta}\ (\neg r) \qquad \frac{\Gamma \Rightarrow \neg\phi,\Delta}{\Gamma,\phi \Rightarrow \Delta}\ (\neg l) \qquad \frac{\Gamma \Rightarrow \phi,\Delta \quad \Gamma \Rightarrow \varphi,\Delta}{\Gamma \Rightarrow \phi\wedge\varphi,\Delta}\ (\wedge r)$$

$$\frac{\Gamma,\phi,\varphi \Rightarrow \Delta}{\Gamma,\phi\wedge\varphi \Rightarrow \Delta}\ (\wedge l) \qquad \frac{\Gamma \Rightarrow \phi[x'/x],\Delta}{\Gamma \Rightarrow \forall x.\phi,\Delta}\ (\forall r) \qquad \frac{\Gamma,\forall x.\phi,\phi[tn/x] \Rightarrow \Delta}{\Gamma,\forall x.\phi \Rightarrow \Delta}\ (\forall l)$$

$$\frac{\Gamma \Rightarrow \phi,\varphi,\Delta}{\Gamma \Rightarrow \phi\vee\varphi,\Delta}\ (\vee r) \qquad \frac{\Gamma,\phi \Rightarrow \Delta \quad \Gamma,\varphi \Rightarrow \Delta}{\Gamma,\phi\vee\varphi \Rightarrow \Delta}\ (\vee l) \qquad \frac{\Gamma,\phi \Rightarrow \varphi,\Delta}{\Gamma \Rightarrow \phi\to\varphi,\Delta}\ (\to r)$$

$$\frac{\Gamma \Rightarrow \phi,\Delta \quad \Gamma,\varphi \Rightarrow \Delta}{\Gamma,\phi\to\varphi \Rightarrow \Delta}\ (\to l) \qquad \frac{\Gamma \Rightarrow \phi[tn/x],\Delta}{\Gamma \Rightarrow \exists x.\phi,\Delta}\ (\exists r) \qquad \frac{\Gamma,\exists x.\phi,\phi[x'/x] \Rightarrow \Delta}{\Gamma,\exists x.\phi \Rightarrow \Delta}\ (\exists l)$$

where Γ,Δ are multi-sets of QF-AFOL formulas.
x' is a new variable w.r.t. Γ,ϕ,Δ, $\phi[tn/x]$ is admissible.

5 A Case Study

In this section, we illustrate how our proposed CDL can be used to capture RTES models and verify CCSL specifications, by analyzing a simple RTES — the Digital Filter (DF) system. The DF system we analyze here is based on [17].

As Fig. 4 shows, the DF is used in a video system, it reads image pixels from a memory, filters them and sends the result out to a video device. The explicit structure of DF is shown in the right figure of Fig. 4. The DF consists of two modules: a Feeder and a Filter. They interact with each other and with their environment through ports p, r and o. Ports are the only way for different modules to communicate in synchronous models. They can be modelled as signal c_p, c_r and c_o in SEP. The behaviour of the DF, as a whole system of two modules, is as follows: the Filter sends a 'Ready' message to the Feeder through port r, to tell it 'I am ready for the pixels'. The Feeder receives this message and the next time it begins feeding pixels towards the Filter, one pixel per unit of time.

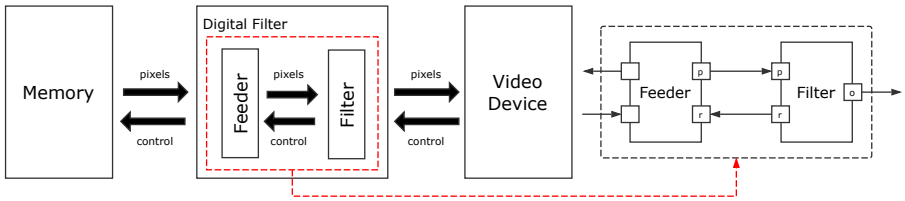


Fig. 4. The Digital Filter system

After the Filter gathers 4 pixels, it runs the computation (instantly) and outputs the result through port o. Then the next time it sends 'Ready' message to the Feeder again....

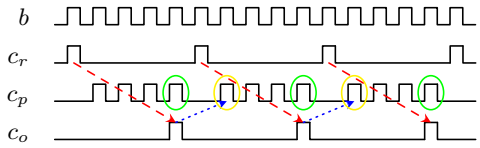


Fig. 5. The schedule of the Digital Filter

The behaviour of the DF can be described by an SEP as follows:

$$DF ::= \alpha_1; (f = 1?\alpha_2; (n = 0 \wedge f = 0?\alpha_3 \cup n > 0 \wedge f = 0?\alpha_4)^*)^\omega,$$

where $\alpha_1 = (c_r | n := 4 | f := 1)$, $\alpha_2 = (c_r | f := 0)$, $\alpha_3 = (c_o | f := 1 | n := 4)$, $\alpha_4 = (c_p | n := n - 1)$. n is for counting the number of pixels the Filter has received. f is a flag, indicating the end of the loop '$(...)^*$'. Fig. 5 shows the schedule of the DF, where clock b is a basic clock. For this DF model we may be interested in two CCSL specifications as follows:

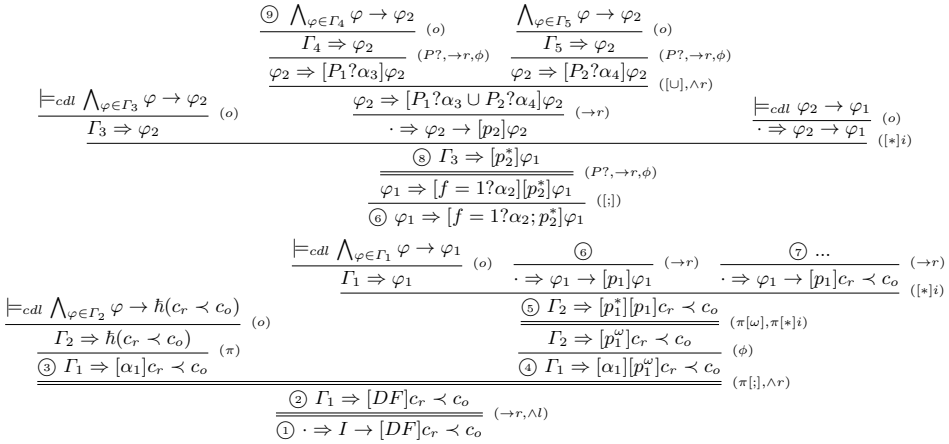
$$SP_1 ::= \langle \{c_r, c_o\}, \emptyset, \{c_r \prec c_o\} \rangle,$$
$$SP_2 ::= \langle \{c_p, c_o, c_{p'}, c_{p''}\}, \{c_{p'} \triangleq c_p \$ 1, c_{p''} \triangleq c_{p'} \propto 3\}, \{c_o \prec c_{p''}\} \rangle.$$

SP_1 expresses the property that 'the result can be obtained only after the "Ready" message is sent', i.e., clock c_r ticks strictly before c_o. SP_2 says that 'only after the last result is computed, the new pixels can be received'. SP_2 contains two clock definitions. $c_{p'}$, $c_{p''}$ are generated clocks not appeared in program DF. Two specifications are indicated by red and blue arrows in Fig. 5 respectively. The ticks of clock $c_{p'}$, $c_{p''}$ are indicated by green and yellow circles respectively.

For SP_1, the verification problem can be captured by a CDL formula:

$$I \rightarrow [DF] c_r \prec c_o,$$

where $I = \bigwedge_{c\in\{c_r,c_p,c_o\}}(c^n = 0 \wedge c^s = 0)$ represents the initial environment of the program. The deduction procedure of this formula is illustrated in Fig. 6. Starting from the root node (node ①), the procedure answers 'yes' iff every leave node of the proof tree returns a valid QF-AFOL formula, which can be checked through an SMT-checking procedure. Inference rules with a 'double line' indicate that there are more than one deductions between the premises and the conclusion, and rules they are based on are listed on the right side. e.g., from node ②, by applying rule $(\pi[;])$, $(\wedge r)$ in sequence, we obtain two premises: node ③ and ④. The meanings of other symbols and contexts are explained in the table of Fig. 6. Note that variables of the form like '$(c_r^n)^1$' is a new general variable (corresponding to c_r^n), not a clock-related variable.



$\alpha_1 = (c_r|n := 4|f := 1)$, $\alpha_2 = (c_r|f := 0)$, $\alpha_3 = (c_o|f := 1|n := 4)$, $\alpha_4 = (c_p|n := n-1)$.
$p_1 = f = 1?\alpha_2; p_2^*$, $p_2 = P_1?\alpha_3 \cup P_2?\alpha_4$, $P_1 = n = 0 \wedge f = 0$, $P_2 = n > 0 \wedge f = 0$.
$\varphi_1 = (C_1 \vee C_2) \wedge c_r^n > c_o^n$, $\varphi_2 = (C_1 \wedge c_r^n - c_o^n > 1) \vee (C_2 \wedge c_r^n > c_o^n)$,
$C_1 = 0 \le n < 4 \wedge f = 0$, $C_2 = n = 4 \wedge f = 1$.
$\boldsymbol{v} = (c_r^n, c_r^s, c_p^n, c_p^s, c_o^n, c_o^s)$, $\boldsymbol{v}^x = ((c_r^n)^x, (c_r^s)^x, (c_p^n)^x, (c_p^s)^x, (c_o^n)^x, (c_o^s)^x)$, where $x = 1, 2, 3, 4$.
$\boldsymbol{v}^x, x_1, x_2, y_1, z_1, z_2, u_1$ are new variables corresponding to their counterparts in substitutions.

$\Gamma_1 : c_r^n = 0, c_r^s = 0, c_p^n = 0, c_p^s = 0, c_o^n = 0, c_o^s = 0$
$\Gamma_2 : \Gamma_1[\boldsymbol{v}^1, x_1, x_2/\boldsymbol{v}, n, 1], c_r^n = (c_r^n)^1 + 1, c_r^s = 1, c_p^n = (c_p^n)^1, c_p^s = 0, c_o^n = (c_o^n)^1, c_o^s = 0, n = 4$
$\Gamma_3 : \{\varphi_1, f = 1\}[\boldsymbol{v}^2, y_1/\boldsymbol{v}, f], c_r^n = (c_r^n)^2 + 1, c_r^s = 1, c_p^n = (c_p^n)^2, c_p^s = 0, c_o^n = (c_o^n)^2, c_o^s = 0, f = 0$
$\Gamma_4 : \{\varphi_2, P_1\}[\boldsymbol{v}^3, z_1, z_2/\boldsymbol{v}, n, 1], c_r^n = (c_r^n)^3, c_r^s = 0, c_p^n = (c_p^n)^3, c_p^s = 0, c_o^n = (c_o^n)^3 + 1, c_o^s = 1, n = 4$
$\Gamma_5 : \{\varphi_2, P_2\}[\boldsymbol{v}^4, u_1/\boldsymbol{v}, n], c_r^n = (c_r^n)^4, c_r^s = 0, c_p^n = (c_p^n)^4 + 1, c_p^s = 1, c_o^n = (c_o^n)^4, c_o^s = 0, n = u_1 - 1$

Fig. 6. The deduction procedure of $I \to [DF]c_r \prec c_o$

Due to the limit of space, we omit the details of the branch from node ⑦. At node ⑤, ⑧, we apply rule $([*]i)$ to eliminate the loop operator $*$. Here we need to manually decide the loop invariants φ_1, φ_2. The selecting of a suitable invariants is according to the loop body (here p_1, p_2) and the formulas we want to verify after the loop program (here $[p_1]c_1 \prec c_2$, φ_1). e.g., in φ_1, we have to guarantee '$c_r^n > c_o^n$' always holds during each execution of p_1, because if not so,

$[p_1]c_1 \prec c_2$ would not hold for some state during the execution of p_1^*. '$C_1 \vee C_2$' is to make sure that n, f can only be 'reasonable values' during the execution of p_1. At last, easy to see that each leave node is a valid QF-AFOL formula. e.g., at node ⑨, clearly from Γ_4, we have C_2 holds. In Γ_4, since $z_1 = 0 \wedge z_2 = 0$(in $P_1[z_1, z_2/n, y]$), there is $(c_r^n)^3 - (c_o^n)^3 > 1$(from $\varphi_1[\boldsymbol{v}^3, z_1, z_2/\boldsymbol{v}, n, f]$). Because $c_r^n = (c_r^n)^3$, $c_o^n = (c_o^n)^3 + 1$, so $c_r^n > c_o^n$ holds.

For SP_2, just like in previous approaches [4, 18, 19], we firstly make the product of the system model DF and the clock definitions $c_{p'} \triangleq c_p\$1$, $c_{p''} \triangleq c_{p'} \propto 3$. As indicated in Fig. 1, this product can then be captured by an SEP program. A similar verification procedure as above can be carried out.

6 Related Work

Previous approaches [18, 19] for the verification of CCSL specifications are mainly based on model checking, where the reachability analysis is made for the product of the system model and the CCSL specification. When the CCSL specification is unsafe, a bound needs to be set to avoid the enumeration of infinite number of states. Our approach is based on theorem proving and SMT-checking, which provides a unified framework under which both safe and unsafe CCSL specifications can be analyzed.

Another subject of analysis for CCSL is to find a schedule of a given CCSL specification [3, 13, 20], where no system models were involved. The earliest approach [3] combined BDD-based boolean solving and the rewriting on clock expressions, while the method in [20] was based on the rewriting logic in Maude. In [13], the schedule was found by solving an UFLIA formula that encodes the CCSL specification through an SMT-checking procedure. Comparing with [13], we propose a proof system to transform the CDL formula into QF-AFOL formulas, which are more efficient for an SMT-checking procedure to solve.

CDL is largely based on the traditional FODL [8] and its rules $(\pi[;])$, $(\pi[\cup])$, $(\pi[*]u)$, $(\pi[*]i)$ are inspired from the Differential Dynamic Temporal Logic (DDTL), a dynamic logic for verification of hybrid systems [16, 21]. In DDTL, the program supports a continuous time model with differential equations embedded into it. Our SEP supports a discrete time model with synchronous mechanism which we think would be more friendly for modelling RTESs.

7 Conclusion and Future Work

In this paper, we propose a logical approach for verification of CCSL specifications. We build a variation of dynamic logic called CDL to capture the verification problem, and a proof system to provide the verification support. We give a case study to illustrate how CDL can be used for verifying CCSL specifications.

Unlike traditional synchronous programming languages, SEP only supports sequential models. We shall present a concurrent extension in a future work by adding a '$\|$' operator. We also consider mechanizing CDL with the popular theorem prover Coq in order to see more practical potentials for this method.

References

1. OMG: UML profile for MARTE: Modeling and analysis of real-time embedded systems. Technical report, OMG (June 2011) formal/11-06-02.
2. Mallet, F.: Clock constraint specification language: specifying clock constraints with UML/MARTE. ISSE **4**(3) (2008) 309–314
3. André, C.: Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA (2009)
4. Mallet, F., de Simone, R.: Correctness issues on MARTE/CCSL constraints. Sci. Comput. Program. **106** (2015) 78–92
5. Mallet, F., Millo, J.V., de Simone, R.: Safe CCSL specifications and marked graphs. In: 11th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign, IEEE (2013) 157–166
6. Zhang, M., Ying, Y.: Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems. In: LCTES '17, ACM (2017) 61–70
7. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. SIGACT News **32**(1) (2001) 66–69
8. Harel, D.: First-Order Dynamic Logic. Volume 68 of LNCS. Springer (1979)
9. Halbwachs, N.: Synchronous programming of reactive systems. Kluwer Academic Pub. (1993)
10. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017) Available at `www.SMT-LIB.org`.
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
12. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
13. Zhang, M., Mallet, F., Zhu, H.: An SMT-based approach to the formal analysis of MARTE/CCSL. In: ICFEM '16, Springer (2016) 433–449
14. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2) (1992) 87–152
15. Gentzen, G.: Untersuchungen über das logische Schließen. PhD thesis, NA Göttingen (1934)
16. Platzer, A.: Logical analysis of hybrid systems - proving theorems for complex dynamics. Springer (2010)
17. André, C., Mallet, F.: Specification and verification of time requirements with CCSL and Esterel. In: LCTES '09, ACM (2009) 167–176
18. Suryadevara, J., Seceleanu, C.C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL mode behaviors using UPPAAL. In: Software Engineering and Formal Methods. Volume 8137 of LNCS., Springer (September 2013) 1–15
19. Zhang, Y., Mallet, F., Chen, Y.: Timed automata semantics of spatio-temporal consistency language STeC. In: TASE '14, IEEE (2014) 201–208
20. Zhang, M., Dai, F., Mallet, F.: Periodic scheduling for MARTE/CCSL: Theory and practice. Sci. Comput. Program. **154** (2018) 42 – 60
21. Platzer, A.: A temporal dynamic logic for verifying hybrid system invariants. In: LFCS '07, Springer (2007) 457–471